대 한 민 국 특 허 청
KOREAN INTELLECTUAL PROPERTY OFFICE

별첨 사본은 아래 출원의 원본과 동일함을 증명함.

This is to certify that the following application annexed hereto is a true copy from the records of the Korean Intellectual Property Office.

출 원 번 호 : 10-2003-0034322
Application Number

출 원 년 월 일 : 2003년 05월 29일
Date of Application MAY 29, 2003

출 원 인 : 주식회사 팬택
Applicant(s) PANTECH CO.,LTD.

2003 년 12 월 04 일

특 허 청
COMMISSIONER

【서지사항】

【서류명】	특허출원서
【권리구분】	특허
【수신처】	특허청장
【참조번호】	0003
【제출일자】	2003.05.29
【발명의 명칭】	이동통신을 위한 임베디드 시스템의 구현 방법
【발명의 영문명칭】	METHOD FOR IMPLEMENTING EMBEDDED SYSTEM FOR MOBILE COMMUNICATION
【출원인】	
【명칭】	주식회사 팬택
【출원인코드】	1-1998-004053-1
【대리인】	
【명칭】	특허법인 신성
【대리인코드】	9-2000-100004-8
【지정된변리사】	변리사 신윤정, 변리사 원석희, 변리사 박해천
【포괄위임등록번호】	2002-089790-8
【발명자】	
【성명의 국문표기】	이명재
【성명의 영문표기】	LEE,Myung Jae
【주민등록번호】	740518-1148622
【우편번호】	152-752
【주소】	서울특별시 구로구 개봉본동 한마을아파트 105-1805
【국적】	KR
【심사청구】	청구
【취지】	특허법 제42조의 규정에 의한 출원, 특허법 제60조의 규정에 의한 출원심사 를 청구합니다. 대리인 특허법인 신성 (인)

【수수료】

【기본출원료】	20	면	29,000 원
【가산출원료】	14	면	14,000 원
【우선권주장료】	0	건	0 원
【심사청구료】	4	항	237,000 원
【합계】	280,000 원		

【첨부서류】 1. 요약서·명세서(도면)_1통

【요약서】

【요약】

본 발명의 이동통신을 위한 임베디드 시스템의 구현 방법은, 호스트에 임베디드 리눅스를 개발하기 위한 환경을 구축하고, 이를 토대로 타겟 보드에서 동작하는 커널과 푸트파일시스템을 이식함으로써, 효율적인 호스트와 타겟 시스템 설정으로 ARM용 크로스 개발환경에 기반하고, 임베디드 리눅스의 어플리케이션 개발시간을 단축시키는 이동통신을 위한 임베디드 시스템의 구현 방법을 제공하는데 그 목적이 있다.

상기 목적을 달성하기 위하여 본 발명은, 타겟 시스템 및 호스트 시스템을 포함하는 시스템에 적용되는 이동통신을 위한 임베디드 시스템의 구현 방법에 있어서, 상기 타겟 시스템에 대한 크로스 개발 환경을 구축하는 개발환경구축단계; 상기 타겟 시스템 및 상기 호스트 시스템의 네트워크 환경을 구축하는 네트워크환경구축단계; 상기 타겟 시스템의 부트 로더를 구성하는 부트로더구성단계; 상기 타겟 시스템에 커널을 구성하는 커널구성단계; 및 상기 타겟 시스템에 GUI 환경을 구성하는 GUI환경구성단계를 포함하고, 상기 커널은 임베디드 리눅스 커널인 것을 특징으로 한다.

【대표도】

도 1

【색인어】

임베디드, 타겟 시스템, 호스트 시스템, ARM

【명세서】

【발명의 명칭】

이동통신을 위한 임베디드 시스템의 구현 방법{METHOD FOR IMPLEMENTING EMBEDDED SYSTEM FOR MOBILE COMMUNICATION}

【도면의 간단한 설명】

도 1은 본 발명의 일 실시예에 의한 이동통신을 위한 임베디드 시스템의 구현 방법을 나타낸 동작흐름도,

도 2는 임베디드 시스템이 구현되는 타겟 시스템의 구성을 나타낸 예시도,

도 3은 커널 이미지(zImage)를 생성하는 루틴을 나타낸 동작흐름도.

【발명의 상세한 설명】

【발명의 목적】

【발명이 속하는 기술분야 및 그 분야의 종래기술】

<4> 본 발명은 이동통신을 위한 임베디드 시스템의 구현 방법에 관한 것으로, 특히, 미리 정해진 특정 기능을 수행하기 위해 컴퓨터의 하드웨어와 소프트웨어가 조합된 임베디드 시스템에 있어서, 다양한 디바이스에 대한 지원성, 안정성, 신속하면서도 정확한 정보처리가 가능한 이동통신을 위한 임베디드 시스템의 구현 방법에 관한 것이다.

<5> 임베디드 시스템(Embedded System)이란 미리 정해진 특정 기능을 수행하기 위해 컴퓨터의 하드웨어와 소프트웨어가 조합된 전자제어 시스템을 말하며, 필요에 따라선 기계의 일부가

포함될 수 있다. 임베디드 시스템에서의 리눅스는 소스가 공개되어 있고 무료로 배포함으로써 가격 경쟁력이 있으며 다양한 디바이스에 대한 지원성, 안정성, 신속하면서도 정확한 정보처리가 가능하다. 또한 임베디드 리눅스는 리눅스 마이크로 커널을 사용할 경우 임베디드 시스템에서의 메모리 사용량을 상당부분 줄일 수 있다. 타겟보드에서 실행될 응용프로그램을 개발하기 위해서는 타겟보드의 기본적인 동작 환경인 루트 파일시스템과 개발환경인 TFTP, BOOTP, NFS등을 구성해주어야 한다. 그러나, 이런 설정들이 응용프로그램 개발과정에 있어서 상당부분 많은 시간을 소비해야 하는 문제점이 있다.

【발명이 이루고자 하는 기술적 과제】

<6> 상기 문제점을 해결하기 위하여 안출된 본 발명은, 호스트에 임베디드 리눅스를 개발하기 위한 환경을 구축하고, 이를 토대로 타겟 보드에서 동작하는 커널과 푸트파일시스템을 이식함으로써, 효율적인 호스트와 타겟 시스템 설정으로 ARM용 크로스 개발환경에 기반하고, 임베디드 리눅스의 어플리케이션 개발시간을 단축시키는 이동통신을 위한 임베디드 시스템의 구현 방법을 제공하는데 그 목적이 있다.

【발명의 구성 및 작용】

<7> 상기 목적을 달성하기 위하여 본 발명의 이동통신을 위한 임베디드 시스템의 구현 방법은, 타겟 시스템 및 호스트 시스템을 포함하는 시스템에 적용되는 이동통

신을 위한 임베디드 시스템의 구현 방법에 있어서, 상기 타겟 시스템에 대한 크로스 개발 환경을 구축하는 개발환경구축단계; 상기 타겟 시스템 및 상기 호스트 시스템의 네트워크 환경을 구축하는 네트워크환경구축단계; 상기 타겟 시스템의 부트 로더를 구성하는 부트로더구성단계; 상기 타겟 시스템에 커널을 구성하는 커널구성단계; 및 상기 타겟 시스템에 GUI 환경을 구성하는 GUI환경구성단계를 포함하고, 상기 커널은 임베디드 리눅스 커널인 것을 특징으로 한다.

<8> 이하, 본 발명이 속하는 기술분야에서 통상의 지식을 가진 자가 본 발명의 기술적 사상을 용이하게 실시할 수 있을 정도로 상세히 설명하기 위하여 본 발명의 가장 바람직한 실시예들을 첨부된 도면을 참조하여 설명하기로 한다.

<9> 도 1은 본 발명의 일 실시예에 의한 이동통신을 위한 임베디드 시스템의 구현 방법을 나타낸 동작흐름도로서, 이에 관하여 설명하면 다음과 같다.

<10> 먼저, 타겟 시스템에 대한 크로스 개발 환경을 구축한다(S101). 크로스 개발환경을 구축하기 위한 각 소프트웨어의 최신버전은 http://www.gnu.org/directory/에서 확인할 수 있다. 여기서 GNU 소프트웨어 목록을 계층적으로 분류해 추적할 수 있다. 소프트웨어의 제작자 및 각 프로젝트의 홈페이지 그리고 원시코드와 문서의 위치를 각 소프트웨어마다 알려주고 있다. 여기서, 주의해야할 사항은 GNU 소프트웨어는 개발초기부터 다중플랫폼을 염두에 두고 있지만 플랫폼별로 패치가 존재할 수 있다는 사실이다. 이러한 패치는 GNU 사이트가 아닌 다른 곳에 있을 가능성이 높기 때문에 찾기 쉽지 않다. 크로스 개발 환경을 구축하려면 GNU 소프트웨어 이외에 커널 원시 코드를 에서 가져와야 한다. 그리고 여기서도 GNU 소프트웨어처럼 패치가 필요하다. 크로스 개발환경을 구축하기 위해서는 필요한 소프트웨어를 먼저 구해야 한다. 원시코드 형식의 소프트웨어의 종류와 내용은 다음과 같다.

<11>

binutils-2.11.2
ftp://ftp.gnu.org/gnu/binutils/binutils-2.11.2.tar.gz
gcc-2.95.3
ftp://ftp.gnu.org/pub/gnu/gcc/gcc-2.95.3.tar.gz
ARM 용 gcc 패치(ver. 2.95.3)
- ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/src/gcc-2.95.3.diff.bz2
glibc-2.2.5
ftp://ftp.gnu.org/gnu/glibc/glibc-2.2.5.tar.gz
ARM 용 glib 패치 (ver. 2.2.5)
- http://embedded-linux.hanbitbook.co.kr/patches/patch-glibc-2.2.5-arm-jhp1
glibc-linuxthreads-2.2.5
ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.2.5.tar.gz
linux-2.4.18
http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.18.tar.gz
커널패치(ARM 용 rmk)
- ftp://ftp.arm.linuxorg.uk/pub/linux/arm/kernel/v2.4/patch-2.4.18-rmk7.gz
커널패치(ARM 용 np)
- ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/v2.4

<12> 시간이 부족하다면 RPM으로 만들어진 이진파일로 크로스 개발 환경을 꾸밀수도 있다. 미리 만들어놓은 이진파일은 다음과 같은 URL에서 구할수 있다.

<13>

ARM 용 크로스 플랫폼 개발환경 패키지
ftp://ftp.netwinder.org/users/c/chagas/arm-linux-cross/RPMS
http://www.lart.tudelft.nl/lartware/compile-tools
ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain

<14> 개발환경 구축에 선행하여, 개발 호스트 플랫폼은 리눅스 운영체제를 탑재한 x86 PC일 수 있고, 임베디드(embedded)는 사용자계정, 작업경로는 /home/embedded/arm 그리고 설치 경로

는 /usr/local/arm-dev/로 정할 수 있으며, 상기 타겟 시스템의 구성은 32bit Intel StrongARM SA1110 RISC Clock 206MHz, 32Mbyte SDRAM, 16Mbyte Flash를 포함할 수 있다. 또한, UART×2, CS8900(10Base-T) 이더넷 컨트롤러(Ethernet Controller), 7.5" STN(640*480) 칼라(color) LCD 기능을 가지고 있는 보드(Hyper 104 SA1110 Evaluation Board)를 사용할 수 있다. 이러한 타겟 시스템의 구성을 나타낸 예시도가 도 2에 도시되어 있다.

<15> 상기 타겟 시스템에 대한 크로스 개발 환경을 구축하는 단계에 있어서, binutils 컴파일과 설치를 위한 입력과정은 하기와 같다.

<16>

```
$ tar xvfz binutils-2.11.2.tar.gz
$ cd binutils-2.11.2
$ ./configure-target=arm-linux-prefix=/usr/local/arm-dev
$ make
$ su - root
Password : *****
# cd /home/embedded/arm/binutils-2.11.2/
# make install
```

<17> 설치가 끝난후 /uer/local/arm-dev에 각종 디렉토리 생성 여부를 확인할 수 있다.

<18> 또한, 상술한 binutils 컴파일과 설치 이후에, gcc 컴파일 및 설치 과정이 수행되는데, gcc 컴파일에 앞서 arm용 리눅스 커널 원시 코드를 풀어야하고, gcc 패키지에 대한 패치작업을 해야한다. 커널 원시코드 압축풀기와 각종패치의 작업은 다음과 같다.

<19>

```
$ tar xvfz linux-2.4.18.tar.gz
$ cd linux
$ zcat ../patch-2.4.18-rmk7.gz | patch -p1
$ cd ../
$ mv linux linux-2.4.18-rmk7
```

<20> gcc 컴파일러를 컴파일하기 전에 커널 헤더 파일 디렉토리를 올바르게 링크시킨다.

<21>

```
$ tar xvfz gcc-2.95.3.tar.gz
$ cd gcc-2.95.3
$ bzcat ../gcc-2.95.3.diff.bz2 | patch -p1
$ cd ..
$ mv gcc-2.95.3 gcc-2.95.3-arm
$ echo "T_CFLAGS = -Dinhibit_libc - D__Gthr_psix_h" >>
  gcc-2.95.3-arm/gcc/config/arm/t-linux
$ su - root
Password: *****
# cd /home/embedded/arm/gcc-2.95.3-amm
# export PATH=' echo $PATH' :/usr/local/arm-dev/bin
# ./configure -target=arm-linux -prefix=/usr/local/arm-dev
  --with-headers=../linux-2.4.18-rmk7/include --disable-
  threads -enable-languages=c
# make
# make install
```

<22> 리눅스 커널의 준비가 끝난후 gcc 초기 컴파일 단계로 들어간다.

<23> 여기서, gcc 초기 컴파일이 필요한 이유는 gcc 교차 컴파일러를 맨처음 컴파일할 때 필요한 환경이 제대로 갖춰져 있지 않기 때문이다. 즉, 타겟 플랫폼과 관련 있는 각종 헤더 파일과 glibc 라이브러리가 없는 상황에서 gcc 크로스 컴파일러의 컴파일이 불가능하기 때문이다.

이런 문제를 극복하는 방법은 처음부터 완벽한 gcc 크로스 컴파일러를 만드는 대신 헤더 파일과 라이브러리를 준비할 수 있는 기능만 탑재한 gcc 부트 스트랩 컴파일러를 만드는 것이다. gcc 2.95.3 계열의 부트 스트랩 컴파일러를 만드는 방식은 다음과 같다.

<24>

```
$ tar xvfz gcc-2.95.3.tar.gz
$ cd gcc-2.95.3
$ bzcat ../gcc-2.95.3.diff.bz2 | patch -p1
$ cd ..
$ mv gcc-2.95.3 gcc-2.95.3-arm
$ echo "T_CFLAGS = -Dinhibit_libc - D__Gthr_psix_h" >>
  gcc-2.95.3-arm/gcc/config/arm/t-linux
$ su - root
Password: *****
# cd /home/embedded/arm/gcc-2.95.3-amm
# export PATH=' echo $PATH' :/usr/local/arm-dev/bin
# ./configure -target=arm-linux -prefix=/usr/local/arm-dev
  --with-headers=../linux-2.4.18-rmk7/include --disable-
  threads -enable-languages=c
# make
# make install
```

<25> gcc 초기 컴파일이 끝나고 뒷부분에서 다룰 glibc 초기 컴파일이 끝나면 gcc 후기 컴파일 과정으로 들어갈 수 있다. gcc 후기 컴파일 과정에서는 부트 스트랩 컴파일 과정에서 설정한 크로스 컴파일 개발 환경을 최대로 사용해 C/C++과 같은 각종 컴파일러를 제대로 만들어 낸다.

<26>

```
$ mv gcc-2.95.3 gcc-2.95.3-bootstrap
$ tar xvpfz gcc-2.95.3.tar.gz
$ cd gcc-2.95.3
$ bzcat ../gcc-2.95.3.diff.bz2 | patch -p1
$ cd ..
$ mv gcc-2.95.3 gcc-2.95.3-arm
$ cd gcc-2.95.3-arm
$ export PATH= 'echo $PATH' :/usr/local/arm-dev/bin
$ ./configure -target=arm-linux --prefix=/usr/local/arm-dev
  --enable-languages=c,c++
$ make all
$ su - root
  Password: *****
# cd /home/embedded/arm/gcc-2.95.3-arm
# export PATH= 'echo $PATH' :/usr/local/arm-dev/bin
# make install
```

<27> 또한, gcc 컴파일과 설치 이후에, glibc 컴파일과 설치 과정을 수행하는데, 부트 스트랩 컴파일러 준비가 끝나면 glibc 컴파일하는 작업을 진행한다. 초기와 후기 컴파일이 필요한 gcc 패키지처럼 glibc도 초기와 후기 컴파일이 필요한데, 이는 타겟 보드에 탑재할 라이브러리를 실행 경로를 바꿔 다시 만들어야 하기 때문이다. glibc 초기 컴파일 과정은 gcc에 비해 상당히 수월한데 그 순서는 다음과 같다.

<28>

```
$ tar xvpfz glibc-2.2.5.tar.gz
$ cat ../patch-glibc-2.2.5-arm-jhp1 | patch -p1
$ cat ../glibc-linuxthreads-2.2.5.tar.gz | tar xvpfz -
$ cd ..
$ mv glibc-2.2.5 glibc-2.2.5-jhp1
$ cd glibc-2.2.5-jhp1
$ export PATH=echo $PATH:/usr/local/arm-dev/bin
$ CC=arm-linux-gcc ./configure arm-linux -prefix=/usr/local/
  arm-dev/arm-linux -enable-add-ons
$ make all
$ su - root
  Password: *****
# cd /home/embedded/arm/glibc-2.2.5-jhp1
# export PATH=' echo $PATH' :/usr/local/arm-dev/bin
# make install
```

<29> gcc 후기 컴파일이 끝나면 glibc 후기 컴파일 작업으로 들어갈 수 있다. Glibc 후기 컴파일 작업에 있어서 make all 후에 make install 명령을 내리면 x86 플랫폼을 위한 glibc 라이브러리를 arm용으로 크로스 컴파일한 glibc 라이브러리로 교체됨으로서 시스템을 다시 설치해야 했었다. 따라서, install_root와 관련된 make 옵션의 충분한 검토가 필요하다. Glibc의 후기 컴파일 과정은 다음과 같다.

<30>

```
$ mv glibc-2.2.5-jhpl glibc-2.2.5-jhpl-bootstrap
$ tar xvfz glibc-2.2.5.tar.gz
$ cat ../patch-glibc-2.2.5-arm-jhpl | patch -p1
$ cat ../glibc-linuxthreads-2.2.5.tar.gz | tar xvfz -
$ cd ..
$ mv glibc-2.2.5 glibc-2.2.5-jhpl
$ cd glibc-2.2.5-jhpl
$ export PATH=' echo $PATH' :/usr/local/arm-dev/bin
$ CC=arm-linux-gcc ./configure arm-linux -prefix=/ --enable
  -add-ons
$ make all
$ su - root
  Password: *****
# cd/home/embedded/arm/glibc-2.2.5-jhpl
# export PATH=echo $PATH:/usr/local/arm-dev/bin
# mkdir /usr/local/arm-dev/target
# mkdir /usr/local/arm-dev/target/glibc-2.2.5
# make install_root=/usr/local/arm-dev/target/glibc-2.2.5
  install
```

<31> 하기와 간단한 프로그램을 하나 컴파일해서 크로스 개발환경 설치를 검증할 수 있다.

<32>

```
#include <stdio.h>
int main()
{
printf( "hello ₩n" );
 }
```

```
$ export PATH=/usr/local/arm-dev/bin:echo $PATH
$ arm-linux-gcc -o hello hello_arm.c
$ file hello
$ arm-linux-readelf -a hello | grep NEEDED
```

<33> file 명령의 결과로서 'hello: ELF 32-bit LSB executable, Advanced RISC Machines ARM, version 1, dynamically linked(uses shared libs), not stripped'와 같은 결과를 확인할 수 있다.

<34> 여기서, 기본적으로 GNU tool이 설치되어 있어야 하며, 크로스 컴파일러는 에서 최신 툴체인(toolchain)을 받아서 설치하였다. 툴체인(toolchain)이란 타겟 디바이스의 S/W 개발을 수행하는데 필요한 호스트 시스템의 크로스 컴파일 환경을 말한다. 툴체인(toolchain)은 여러 소스들을 컴파일하고 빌드(build)하여 실행파일을 생성하는데 필요한 각종 유틸리티 및 라이브러리의 집합체이다. 여기서는 GNU에서 제공하고 있는 GNU C, C++용 GCC compiler와 GNU binary utilities 그리고 GNU C library를 사용하였다. 또한, StrongARM에 사용하기 위한 ARM용 toolchain은 다음과 같이 호스트에 설치한 RPM 패키지이다.

<35>

arm-linux-binutils-2.10-1.i386.rpm

arm-linux-gcc-2.95.2-2.i386.rpm

arm-linux-glibc-2.1.3-2.i386.rpm

<36> 그 후, 네트워크 환경을 구축한다(S102). 이러한 네트워크 환경의 구축은 TFTP, BOOTP/DHCP 설치와 환경설정, NFS 환경설정 미니컴(minicom) 설정으로 나눌수 있다.

<37> Bootp는 TCP/IP상에서 자동부팅을 위한 최초의 표준으로써 diskless system이 부팅시 IP를 포함한 System configuration을 설정하는 방법이며 UDP와 TFTP를 사용한다. 이때, 이더넷(ethernet)을 이용한 tftp 방법이 상당히 시간절약 및 에러방지에 효율적이다. 리눅스(Linux)에서의 TCP/IP 통신은 inetd를 통해 이루어지며, 타겟보드는 호스트 시스템에 IP를 요청하면 호스트 시스템은 IP를 부여하게 된다.

<38> Bootp 프로세스는 자신의 구성 화일을 읽어서 해당 MAC 어드레스(address)를 가진 항목이 있는가를 확인하고 항목이 존재한다면 들어있는 정보를 가지고 응답 패킷을 구성하게 되며, bootptab에 설정된 정보가 타겟 보드로 전송되어진다. Tftp는 ftp와 네트워크를 통한 파일 전송서비스라는 점은 비슷하지만 ftp는 tcp 전송방식을 사용하고 tftp는 udp를 통한 단방향 핸드쉐이킹 전송방식을 사용한다. upd는 신뢰성이 부족하지만 구조가 상당히 단순화되어서 빠른 전송이 가능하다. upd는 단지 메시지를 broadcast를 하게되어 타겟보드에서 수신하는지에 상관없이 타겟보드를 향해 데이터를 보낸다.

<39> 네트워크 구축 접근 방법을 상세히 살펴보면, 먼저, /etc/xinetd.d/tftp 파일을 다음과 같이 생성한다.

<40>

```
# cat >> /etc/xinetd.d/tftp <<  "EOF"
service tftp
{
    disable = no
    socket_type = dgram
    protocol = udp
    wait      = yes
    user      = root
    log_on_success  +=USERID
    log_on_failure  +=USERID
    server           = /user/local/libexec/tftpd
    server_args      = /tftpboot
}
EOF
```

<41> /etc/rc.d/init.d/xinetd를 재시작함으로써 환경설정파일을 다시 읽게한 후 GNU에서 나온 inetutils 패키지에 들어있는 tftpd를 다음과 같은 방법으로 컴파일한 후 설치한다.

<42>

```
$ tar xvpfz inetutils-1.4.0.tar.gz
$ ./configure
$ make all
$ su - root
# cd /home/embedded/network/inetutils-1.4.0
# make install
```

<43> 테스트를 위해 tftp 최상위 디렉토리를 만들고 다음과 같은 테스트 파일을 생성한다.

<44>

```
# mkdir /tftpboot
# cat >> /tftpboot/test.txt << "EOF"
  tftp tset
  EOF
```

<45> 개발 호스트에서 다음의 명령으로 tftp가 동작하는지 확인한다.

<46>

```
% tftp localhost
tftp> get test.txt
tftp > quit
```

<47> 이제 bootp/DHCP 설치와 환경설정을 생각해보자. 만약, DHCP 서버를 설치하지 않았다면 다음과 같은 방법으로 DHCP 서버를 먼저 설치한다.

<48>

```
$ tar xvpfz dhcp-latest.tar.gz
$ cd dhcp-3.0p11
$ ./configure
$ make all
$ su - root
  Password: *****
# cd /home/embedded/network/dhcp-3.0p11
# make install
```

<49> 환경설정부분은 /etc/dhcp.conf 파일을 만들거나 이미 존재하면 필요한 부분을 추가/수정한다. 동일한 서브넷에 BOOTP/DHCP 서버를 둘이상 설치하면 클라이언트가 혼란을 일으킬 수 있으므로, BOOTP/DHCP 서비스는 반드시 호스트 하나가 담당하게 한다.

<50> 그 후, /var/state/dhcp/dhcpd.leases이나 /var/lib/dhcp/dhcpd.leases 파일을 확인한 후 해당 파일을 다음과 같이 생성하고 dhcpd를 시작한다.

<51>

```
# touch /var/state/dhcp/dhcpd.leases
```

<52> 설정이 모드 끝나면 동작 및 확인은 다음과 같이 할 수 있으며,

<53>

```
# netstat -a | grep bootps
```

<54> 다음과 같은 결과출력을 볼 수 있다.

<55>

```
udp    0      0 *:bootps                *:*
```

<56> 실제 구현과정에서 본 타겟보드로의 이미지 다운로드 중에 데이터의 유실이 많이 발생하였는데 파일이미지가 클수록 유실확률이 큰 것을 볼 수 있다. 본 구현중의 성공확률은 약 20% 정도였으며 호스트 시스템의 하드웨어 사양이나 설정방법에 따라 유동적인 모습을 나타내었다. 다음은 호스트환경에서 설정한 /etc/bootptab 및 /etc/inetd.d/tftp의 설정 정보이다.

<57>

```
Target board:₩
ht = 1:₩
ha = 0x00d0caf12611:₩
ip = 166.104.115.151
sm = 255.255.255.0

Target board : label
 ht : hardware type(1= Ethernet)
 ha : hardware address
  ( address 와 target board 의 MAC address 가 같아야 한다.)
 sm : subnet mask

 Service tftp
    {
          disable = no
          socket_type =dgram
          protocol = udp
          wait = yes
          user = root
          server = /usr/sbin/in.tftpd
          server_args = -s /tftp
 }
```

<58> /etc/export 파일 편집 또는 생성 후 공유 디렉토리를 외부로 엑스포트(export)시킨다. 또한, NFS 데몬이 앞서 지정한 디렉토리를 외부로 엑스포트 하도록 exportfs -a 명령을 내린다. 마지막으로 /etc/rc.d/init.d/nfs restart하면 된다.

<59> 개발 호스트에서의 동작확인은 서버와 클라이언트쪽에서 아래와 같이 각각 확인하면 된다.

<60>

```
# ps -ef | grep nfsd
# netstat -a | grep nfs
# showmount -e localhost
```

<61> 임베디드 리눅스를 탑재한 타겟 보드는 콘솔을 사용하기 위해 대부분 직렬 포트를 통해 개발 호스트에 연결하게 된다.

<62> 그 후, 부트 로더를 구성한다(S103). 즉, 메모리에 적재하여 부팅하는 방법으로 설계하고, 이 이미지를 플래시 메모리에 저장하여 부트 ROM의 기능을 하게 된다. BIOS가 주축이 되는 x86환경과 달리 ARM 환경에서는 처음 전원이 들어오면 CPU가 호출하는 초기 엔트리 포인트 위치에 부트 로더를 위치시켜야 한다. 그리고 이렇게 플래시 메모리에 부트 로더를 구워주는 기능을 JTAG 소프트웨어가 담당한다.

<63> JTAG 소프트웨어를 컴파일하기 전에 하드웨어 요구사항을 점검한다. 타겟 보드에는 연결 단자가 보통 세 개 있다. 하나는 콘솔 입출력을 위한 직렬 단자이고, 다른 하나는 JTAG를 위한 병렬 단자, 마지막 하나는 네트워크를 위한 RJ-45 단자이다. 부트 로더를 플래시 메모리에 굽기 위해서는 타겟보드에 있는 JTAG 단자와 개발 호스트에 있는 병렬 포트 단자 사이에 반드시 케이블을 연결해야 한다. JTAG 소프트웨어는 JTAG용 하드웨어가 동일하지 않으면 동작시킬 수 없으므로 개발보드 회사에서 제공하는 소프트웨어를 쓰는 것이 무난하다.

<64> 부트로더의 이미지(blob)는 LART()에서 제공하는 blob을 JTAG를 이용한 플래시 퓨징(flash fusing) 방식으로 다운로드할 수 있고, H/W의 초기화, 리눅스의 부팅, 커널 혹은 램디스크 다운로드, 다운로드한 커널 및 램디스크를 플래시(flash)에 쓰기, TFTP를 통한 SDRAM에

다운로드가 가능하다. 또한, 호스트 시스템(host system)에서 동작하는 jflash라는 프로그램을 사용하였으며, 이 jflash는 병렬 포트(parallel port)를 이용하여 타겟 보드(target board)에서 필요로 하는 JTAG 신호를 생성한다. 호스트 시스템(host system)에서 생성된 JTAG 신호는 병렬 케이블(parallel cable)을 통해 동글(dongle)에 전달된다. 동글(Dongle)은 TTL 74HCT541을 사용하여 구현할 수 있으며, 이 기능은 단지 호스트 시스템(host system)의 병렬 포트(Parallel Port)에서 발생한 5V 전압을 SA1110에 적합한 3.3V 전압 레벨로 변환하는 기능이다.

<65> 동글(Dongle)을 통해서 전달된 JTAG 신호 중 TMS와 TCLK는 TAP에 전달되어 JTAG의 상태 머신(state machine)을 결정하고, TDO와 TDI는 JTAG의 명령의 상태에 따라 바이패스 레지스터(bypass register), 바운더리 스캔 셀(boundary scan cell), ID 레지스터(register) 등의 입력과 출력 부분에 연결된다. SA1110의 JTAG을 통해서 플래시(flash) 메모리가 필요로 하는 버스 타이밍(bus timing)을 발생하여 플래시(flash)에 전달한다.

<66> 호스트 시스템(host system)의 쉘(shell)에서 jflash blob을 입력하면 blob의 이진 코드(binary code)가 플래시(flash) 메모리의 0번지부터 퓨징(fusing)을 한다. 메모리에 쓰기 전에 기본적으로 쓰고자 하는 0번 블록(block)을 지우고, blob을 퓨징(fusing)한 후, 에러 없이 퓨징(fusing)되었는지를 검사하기 위해 검증을 한 후에 이상이 없으면 blob이 정상적으로 로딩(loading)을 완료하게 된다. 이 상태에서 직렬(serial) 통신 환경의 설정에 이상이 없다면 정상적으로 blobdl 동작하는 것을 볼 수 있다.

<67> 먼저 start.S를 통해 각종 하드웨어 초기화가 이루어지고 start.S에서 main.c의 c_main() 펑션으로 점프를 한다. c_main()에는 먼저 시리얼 및 타이머를 초기화하고 다음 커널(kernel)과 램 디스크(ram disk)를 sdram으로 리로딩(reloading)을 하고 커맨드를 기다린다.

<68> 만약, 아무 커맨드가 없으면 bootkernel() 함수를 통하여 바로 커널(kernel)이 실행되고 커맨드가 있다면 부트로더 명령을 할 수 있게 프롬프트가 떨어진다. 프롬프트상에서 명령을 기다리고 있다가 명령이 들어오면 GetCommand() 함수를 통해 명령을 분석하여 해당 펑션이 호출되어 해당 명령이 실행된다.

<69> 각종 명령은 bootkernel() 함수를 통해 커널이 부팅되고 download() 함수를 통해 호스트(Host)로부터 시리얼로 데이터가 sdram 영역으로 다운로딩이 되고 플래시(flash) 등등 여러 가지 기능이 수행된다.

<70> Blob에 의한 부팅 과정은 터미널 에뮬레이터인 미니컴(minicom)으로 호스트에서 관찰할 수 있으며, 터미널에 대한 세팅은 115200 baud, 8 data bits, no parity, 1 stop bit, no start bits로 하였다.

<71>

```
Consider yourself LARTed!

blob version 2.0.5-pre2 for Hyper104
Copyright (C) 1999 2000 2001 Jan-Derk Bakker and Erik Mouw
blob comes with ABSOLUTELY NO WARRANTY; read the GNU GPL for details.
This is free software, and you are welcome to redistribute it
under certain conditions; read the GNU GPL for details.....
Memory map:
  0x02000000 @ 0xc0000000 (32 MB)
ELF sections layout:
  0xc0200400 - 0xc0206694 .text
  0xc0206694 - 0xc02076ff .rodata
  0xc0207700 - 0xc0207cc6 .data
  0xc0207cc8 - 0xc0207cc8 .got
  0xc0207cc8 - 0xc0207e1c .commandlist
  0xc0207e1c - 0xc0207e7c .initlist
  0xc0207e7c - 0xc0207e88 .exitlist
  0xc0207e88 - 0xc0207eb8 .ptaglist
  0xc0207ec0 - 0xc020af68 .bss
  0xc0208f68 - 0xc020af68 .stack (in .bss)
Loading blob from flash . done
Loading kernel from flash ... done
Loading ramdisk from flash ..... done
Autoboot in progress, press any key to stop ...
Starting kernel ...
```

<72> blob의 설치는 다음과 같은 순서대로 진행한다.

<73> 1. blob을 받아서 압축을 푼다.

<74> 2. $ tar xzvf blob.tar

<75> 3. 크로스 컴파일러(Cross Compiler)를 설치할 때와 마찬가지로 /root의 .bash_profile profile을 편집한다.

<76>
```
CC=armv4l-unknown-linux-gcc
OBJCOPY=armv4l-unknown-linux-objcopy
Export CC OBJCOPY
```

<77> 4. 다음과 같이 실행하면 편집한 bash_profile을 사용할 수 있다.

<78> $ source .bash_profile

<79> 5. 이후에 압축을 푼 폴더로 간다.

<80> 6. 아래와 같은 명령어를 실행하면 config가 자동 설정한다.

<81>
```
./configure --with-linux-prefix=/usr/local/arm/armv4l- unknown-linux --with-board=assabet arm-assabet-linux-gnu
```

<82> 컴파일이 에러없이 수행되면 src/에 blob이라는 이진(binary) 파일이 생긴다. 이 파일을 jflash를 이용하여 타겟보드의 플래시(flash) 영역에 다운로드한다.

<83> 타겟시스템의 플래시(flash) 메모리 영역의 시작번지에는 부트로더가 있으며, 타겟시스템이 동작하기 시작할 때의 시작번지가 부트로더 영역이다.

<84> 그 후, 커널을 구성한다(S104). 커널 컴파일하는 과정은 x86 계열의 커널 컴파일하는 과정과 크게 다르지 않다. 커널구성은 make config, make menuconfig, make xconfig 등이 있다. 이중에서 가장 많이 사용되어지는 lxdialog를 이용하여 make menuconfig를 사용할 수 있으며, make dep; make zImage; make modules; make modules_install 과정순으로 진행한다. make zImage로 생성된 커널이미지는 arch/arm/boot에 존재하며 새로 생성된 zImage를 /tftp로 복사하여 부트로더에서 다운로드한다. 커널 이미지(zImage)는 도 3에 도시된 바와 같은 루틴에 의해 만든다.

<85> 타겟 보드는 두가지의 구성(configuration)이 존재한다.

<86> 첫째는 램 디스크(ramdisk)를 사용하는 방법이고 둘째는 cramfs를 루트 파일 시스템(root filesystem)으로 사용하는 방법이다. cramfs는 ROM 파일시스템(filesystem)으로 읽기 전용(read-only)으로만 사용가능하며 이 또한 데이터의 손실을 막을 수가 있는 장점이 있다. cramfs는 압축 알고리즘(gzip)을 사용하기 때문에 그만큼 용량을 줄일 수 있으며 플래시(Flash)에서 실행되기 때문에 램 디스크(ramdisk)에 비해 램의 사용량을 줄일 수가 있다. 이것을 플래시(flash) 메모리에 사용하였다.

<87> 그 후, GUI 환경을 구성한다(S105). 임베디드 리눅스(Embedded linux) 환경에서의 이동통신을 위한 어플리케이션 개발은 GUI 툴키트(toolkit)의 선택이 상당히 중요한데, 타겟 보드의 포팅(porting)에 사용되어지는 GUI 툴키트(toolkit)로서 Qt/embedded를 사용할 수 있다. 여기서, 상기 Qt/embedded는 X-window 없이 리눅스(linux) 커널이 제공하는 프레임 버퍼(Frame Buffer)를 이용하는 방식을 사용한다. QT/embedded를 구현하는 과정은 하기와 같다.

<88>
```
# mv qt-embedded-2.3.2.tar.gz /usr/local/
# tar xvfz qt-embedded-2.3.2.tar.gz
# mv qt-2.3.2 qte-2.3.2
# cd qte-2.3.2
# vi INSTALL
```

<89> INSTALL 파일 내용을 확인하고, 자신의 환경설정에 맞게 수정한다.

<90> 또한, .bash_profile 환경설정파일에 다음부분을 추가한다.

<91>
```
export QTDIR=/usr/local/qte-2.3.2
export LD_LIBRARY_PATH=/usr/local/qte-2.3.2/
        lib:$LD_LIBRARY_PATH
* QTDIR- Qt 가 설치되어있는 디렉토리
* Qt 에서 사용하는 공유 라이브러리가 있는 디렉토리
```

<92> 추가가 끝났으면 #./configure한 후 하기의 과정을 거치면 Qt/embedded의 설치가 끝난다.

<93>
```
1. 라이센스 동의 질문에 yes 입력하고 엔터
2. feature configuration 설정에서 Everything 선택 : 5 번
3. color depth 선택에서 16bpp 선택 : 16
4. Qt Virtual Framebuffer 지원여부선택 : yes
# make
```

<94> /usr/local/qte-2.3.2/lib에서 라이브러리를 다음과 같이 확인하면 된다.

<95>

```
libqte.so -> libqte.so.2.3.2
libqte.so.2 -> libqte.so.2.3.2
libqte.so2.3 -> libqte.so.2.3.2
libqte.so.2.3.2
```

<96> 위의 네 개의 파일이 존재하는지 확인하고 다시 로그인해서 #echo$QTDIR로 확인한다. 경로명은 /usr/local/qte-2.3.2일 수 있다. Qt Virtual FrameBuffer를 사용하기 위해서는 커널 컴파일 옵션에서 필요한 항목들을 설정한 후에 다시 커널 컴파일을 해야 한다.

#cd/usr/src/linux 혹은 cd/usr/src/linux-2.X.X(커널이 존재하는 디렉토리)로 이동한 후 # make menuconfig한다.

<97>

1. 첫번째 항목인 Cide maturity level options 선택한후 세부 설정항목에서 Prompt for development and/or incomplete code/drivers 항목을 스페이스바를 이용해 선택.
2. Exit 선택후 초기 메뉴에서 Console Drivers 선택한후 세부 설정항목에서 Frame-buffer support 항목을 선택.
3. 다시 세부항목에서 Virtual Frame Buffer support 선택하고 다시 Advance low level driver options 에서 8bpp packed pixels support 와 16bpp packed pixels support 선택.
4. Exit 한후 저장하고 빠져나온다.

<98> 여기까지 커널메뉴에서의 항목설정이다.

<99> # make dep; make clean; make; make install; make modules; make modules_install 순서로 커널 컴파일을 다시 한다. Qt Virtual FrameBuffer를 사용하기 위해서는 x86에 설치된 리

눅스에서 사용하여 개발해야 하기 때문에 Qt/X11용으로 컴파일된 Qt Virtual FrameBuffer를 설치해야 한다.

<100> Source 주소는 ftp://ftp.trolltech.com/qt/source/qt-x11-2.3.2.tar.gz이다. 물론 그 이상의 버전을 받아도 무관하다. 계정을 하나 만들고(user는 embedded로 하였다.) 임베디드(embedded) 계정으로 로그인한다. 압축을 푼 후 임베디드(embedded) 홈 디렉토리에서 (cd/home/embedded).bash_profile에서 다음과 같이 추가한다.

<101>

```
export QTDIR=~/qtx-2.3.2
export LD_LIBRARY_PATH=~/qtx-2.3.2/
        lib:$LD_LIBRARY_PATH
```

<102> 환경설정이 끝난 후 #./configure 실행한 후 라이센스 동의에 'yes'하고 컴파일이 끝난 후 qvfb 컴파일을 한다. 이제 qvfb를 실행한다.

<103>

```
# ./qvfb -width 640 -height 480 -depth 16 &
```

<104> x86에 리눅스를 설치하고 개발에 필요한 GUI 툴키트(Toolkit)인 Qt/Embedded를 설치하였고 임베디드 시스템(embedded system)의 에뮬레이터로 사용할 수 있는 Qt 가상 프레임 버퍼(Virtual FrameBuffer)를 설치할 수 있다. Qt 가상 프레임 버퍼를 사용하기 위해선 Qt/X11이 필요하므로 설치한다. 또한, .bash_profile에서 다음과 같이 추가한다.

<105>

```
export PATH=/home/embedded/qtx-2.3.2/bin:$PATH
```

<106> #cp~/qtx-2.3.2/tools/qvfb/qvfb/qvfb ~/qtx-2.3.2/bin으로 복사해서 어느 디렉토리에서든지 사용가능하게 하였다. Qt 가상 프레임 버퍼 테스트는 루트(root)로 로그인한 후 아래와 같이 하면 된다.

<107>

```
# ./qvfb -width 640 -height 480 -depth 16 &
Virtual FrameBuffer 띄우기
# cd $QTDIR
# cd examples/launcher/
# ./launcher -qws
```

<108> 이상에서 설명한 본 발명은, 본 발명이 속하는 기술분야에서 통상의 지식을 가진 자에 있어 본 발명의 기술적 사상을 벗어나지 않는 범위 내에서 여러 가지로 치환, 변형 및 변경이 가능하므로 전술한 실시예 및 첨부된 도면에 한정되는 것이 아니다.

【발명의 효과】

<109> 본 발명은 호스트에 임베디드 리눅스를 개발하기 위한 환경을 구축하고, 이를 토대로 타겟 보드에서 동작하는 커널과 푸트파일시스템을 이식함으로써, 효율적인 호스트와 타겟 시스템 설정으로 ARM용 크로스 개발환경에 기반하고, 임베디드 리눅스의 어플리케이션 개발시간을 단축시키는 장점이 있다.

【특허청구범위】

【청구항 1】

타겟 시스템 및 호스트 시스템을 포함하는 시스템에 적용되는 이동통신을 위한 임베디드 시스템의 구현 방법에 있어서,

상기 타겟 시스템에 대한 크로스 개발 환경을 구축하는 개발환경구축단계;

상기 타겟 시스템 및 상기 호스트 시스템의 네트워크 환경을 구축하는 네트워크환경구축단계;

상기 타겟 시스템의 부트 로더를 구성하는 부트로더구성단계;

상기 타겟 시스템에 커널을 구성하는 커널구성단계; 및

상기 타겟 시스템에 GUI 환경을 구성하는 GUI환경구성단계

를 포함하고,

상기 커널은 임베디드 리눅스 커널인

것을 특징으로 하는 이동통신을 위한 임베디드 시스템의 구현 방법.

【청구항 2】

제1항에 있어서, 상기 개발환경구축단계는,

gcc 부트 스트랩 컴파일러에 의해 gcc 초기 컴파일 과정을 수행하는 단계

를 포함하고,

상기 gcc 부트 스트랩 컴파일러는 헤더 파일 및 라이브러리를 준비하는 기능을 탑재한 것을 특징으로 하는 이동통신을 위한 임베디드 시스템의 구현 방법.

【청구항 3】

제1항에 있어서, 상기 네트워크환경구축단계는,

이더넷을 이용한 TFTP 방식에 따라 상기 타겟 시스템 및 상기 호스트 시스템의 네트워크 환경을 구축하는 단계인

것을 특징으로 하는 이동통신을 위한 임베디드 시스템의 구현 방법.

【청구항 4】

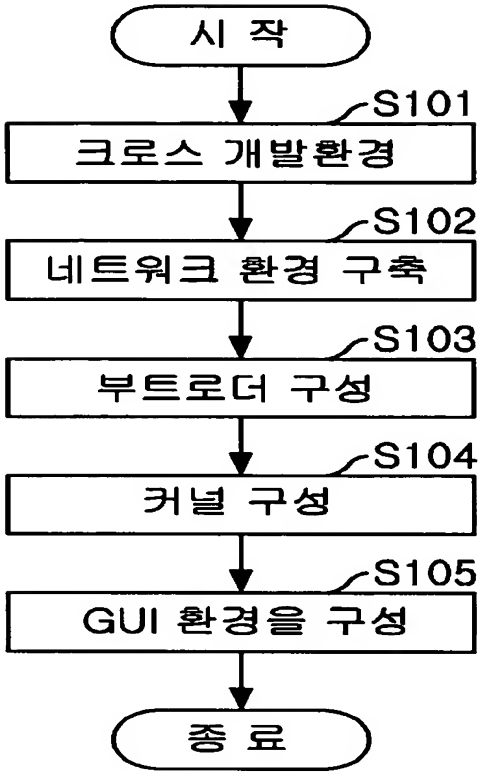
제1항에 있어서, 상기부트로더구성단계는,

읽기 전용 파일 시스템을 루트 파일 시스템으로 사용하여 상기 타겟 시스템의 부트 로더를 구성하는 단계인

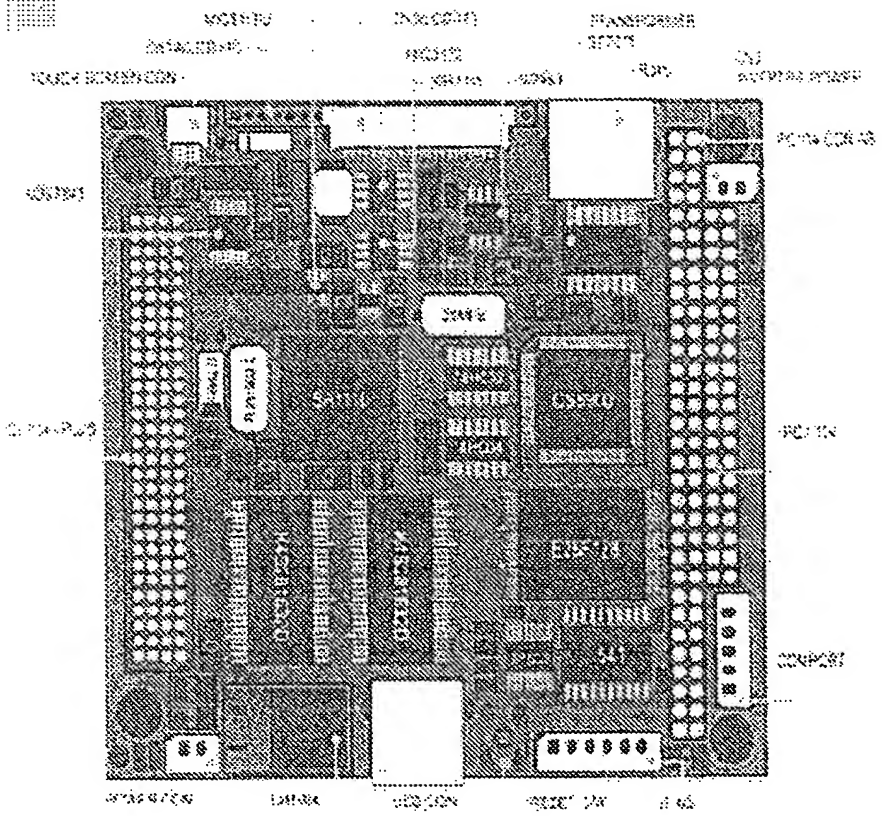
것을 특징으로 하는 이동통신을 위한 임베디드 시스템의 구현 방법.

【도면】

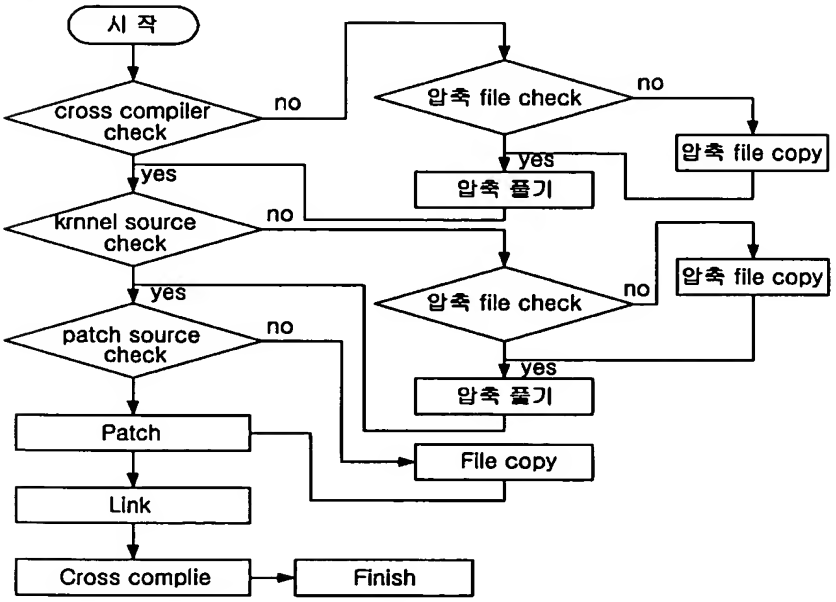
【도 1】


시 작
S101
크로스 개발환경
S102
네트워크 환경 구축
S103
부트로더 구성
S104
커널 구성
S105
GUI 환경을 구성
종 료

【도 2】



【도 3】


시 작
cross compiler check
no
yes
krnnel source check
no
yes
patch source check
no
yes
Patch
Link
Cross complie
Finish
압축 file check
no
yes
압축 file copy
압축 풀기
압축 file check
no
yes
압축 file copy
압축 풀기
File copy